TESTE DE *SOFTWARE*

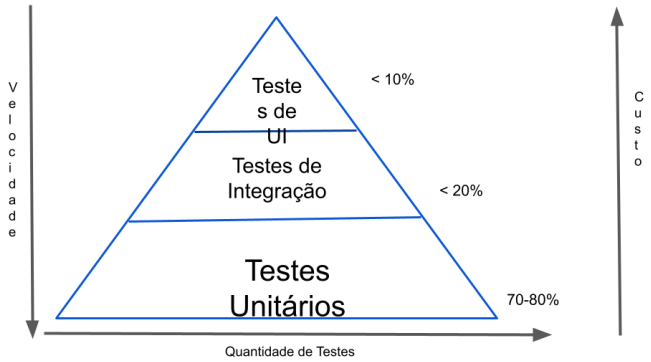
AULA 4

Prof.ª Maristela Weinfurter

CONVERSA INICIAL

A visão sistêmica em testes de *software* nos remete ao desenho da pirâmide de testes, segundo a Figura 1.

Figura 1 – Pirâmide de testes



Os testes unitários formam a base da pirâmide, com os testes de integração dela e os testes de interface do usuário no pico. Testes unitários são pequenos e isolados, que cobrem funcionalidades limitadas, enquanto os testes de integração juntam os testes unitários, agora dentro de uma funcionalidade interdependente.

Os principais pontos fortes dos testes unitários são o quão pequenos e rápidos eles são para executar. Como cada teste é muito limitado, precisaremos muitos deles, mas, tudo bem, porque eles são rápidos de escrever e executar.

No próximo nível, estão os testes de integração. Eles cobrem uma ampla gama de funcionalidades, mas ainda estão limitados a uma área do sistema. Devemos tentar ter menos deles, pois são mais difíceis de escrever, executar e manter.

Finalmente, no nível superior, estão os testes do sistema. Estes são os mais lentos de executar e os mais difíceis de manter, mas fornecem uma cobertura inestimável de todo o sistema trabalhando em conjunto. Pode haver menos deles, mas eles nos dão mais confiança de como seu produto funcionará na prática, devido ao seu escopo e à sua complexidade.

Todas as classificações ou categorizações nas quais os vários tipos de testes se encontram fazem parte de tal visão sistêmica. Isso nos remete à ideia de que o caminho de testes que garanta a qualidade de nosso *software* deve ser bem planejado e adaptado à realidade do nosso projeto e da nossa empresa.

TEMA 1 – CASOS DE TESTES

Casos de testes são importantes dentro do processo de teste de *software*. Eles descrevem determinada condição que deve ser testada através de um conjunto de valores de entrada, restrições para que a execução ocorra e o resultado ou comportamento esperado.

Segundo Amey (2022), cada teste compreende quatro elementos:

1. pré-requisitos;
2. configurar;
3. procedimento;
4. resultado.

Primeiramente, é necessário configurarmos os pré-requisitos necessários: estamos executando os serviços corretos com a versão e configuração corretas? É óbvio, mas é fácil perder uma etapa crítica e perder tempo de teste.

A próxima etapa é a definição da versão, configuração e ambiente corretos, além de descrever os pré-requisitos do teste com mais detalhes. Estes só precisam ser preparados uma vez para toda uma série de testes.

Para cada teste, precisamos ter certeza de que a configuração está correta. É importante termos certeza de que um usuário ou um determinado conjunto de dados ou o requisito com informação foram considerados no estado inicial necessário para o teste e torná-lo explícito. O truque é deixar as suposições claras. Há um número infinito de variáveis que podemos especificar; precisamos escolher as variáveis relevantes para este teste. Enquanto alguns são óbvios, outros podem não ser. Os usuários precisam limpar seus *cookies*, desinstalar programas ou limpar as entradas do registro antes de um teste. Que estado pode ser deixado por testes anteriores que podem confundir seus testes? São algumas preocupações que podemos responder através de perguntas ou então através de um *checklist*.

Não são raros casos que são testadas muitas compilações de desenvolvimento durante o desenvolvimento, localizando e corrigindo dezenas de *bugs*. Quando finalmente o *deploy* em produção é feito, tudo começa a falhar imediatamente, havendo a necessidade de reverter (*rollback*). Apesar do

ocorrido, o fato é que nunca havíamos tido esse problema em nenhum de nossos testes, mas todas as atualizações em produção falharam. O que deu errado?

Na realidade, o sistema em produção saltou da última versão anterior, 5.0.17, para a nossa nova versão, 5.1.23. No entanto, ele não passou por todas as versões intermediárias de 5.1.1, 5.1.2, 5.1.3, e assim por diante. Uma de nossas migrações de banco de dados estava com defeito e só funcionaria se fosse realizada em etapas. Atualizamos gradualmente todos os nossos sistemas de teste entre essas versões, mas o sistema em produção não. Precisávamos de um novo bloqueio de novos testes, obrigando-nos a realizar atualizações garantindo a sequência exata de atualizações no ambiente de produção. Em outras palavras, exigimos um pré-requisito explícito para não ignorar compilações intermediárias.

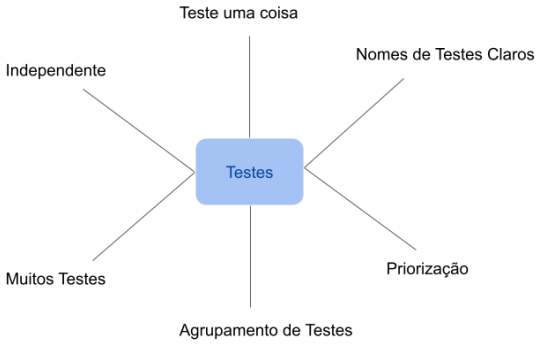
Com o procedimento e a configuração claramente descritos, podemos realizar os testes planejados. Finalmente, e não menos importante, é o momento de documentar o resultado. Assim como na configuração, há muitos estados diferentes que devemos verificar. A habilidade em projetar os testes é escolher cuidadosamente as verificações que realizaremos.

1.1 Determinando o que verificar

É necessário documentar cada parte do caso de teste, portanto, precisamos de um sistema de gerenciamento de teste para rastreá-los. Além de ter etapas explícitas para cada teste, precisamos fazer referência a elas. Todo teste deve ter um ID vinculado aos requisitos que cobre e aos *bugs* que revela. Isso permite rastrear quais testes foram mais úteis.

Cada teste deve seguir os seis princípios da Figura 2

Figura 2 – Princípios no projeto de escrita de casos de testes



Fonte: elaborado com base em Amey, 2022.

- Independência de outros testes (Independente): idealmente, cada teste passaria ou falharia por conta própria. Dessa forma, onde houver uma falha, haverá um indicador claro, em vez de muitos testes falharem de uma só vez. Na prática, isso é difícil de conseguir, e muitos testes dependem da aprovação em testes anteriores. Nesse caso, é importante fazer os testes de forma o mais linear possível, assim fica óbvio que nada além de uma determinada etapa funcionou. Novamente, isso torna a investigação muito mais fácil.
- Cada teste deve verificar uma coisa (Teste uma Coisa): além de ser independente dos outros, cada teste deve ter um propósito e verificar apenas uma parte da funcionalidade, na medida do possível. Então, sabemos a causa da falha e podemos facilmente depurá-la.
- Não ter medo de fazer muitos testes (Muitos Testes): um corolário para cada teste verificando apenas uma coisa é o que precisamos de muitos deles. Novamente, tudo bem. Disponha-os claramente.
- Agrupar seus testes (Agrupamento): com muitos testes para gerenciar, precisamos verificar se eles estão divididos em diferentes seções e pastas para que saibamos onde encontrá-los e onde novos devem ser adicionados.
- Fornecer nomes claros aos testes (Nomes Claros): ao examinar uma lista de falhas, devemos facilitar a identificação do que deu errado ao incluirmos nomes claros. Isso pode significar que precisamos de nomes longos, e tudo bem.
- Priorizar os casos de teste (Priorização): planejar para termos muitos casos de teste, mas não que todos sejam iguais. Se eles começarem a demorar muito para serem executados, concentre-se nos testes de caminho feliz e nas áreas de fraqueza conhecidas, onde encontramos *bugs* antes.

Seguir esses princípios também torna as estatísticas de teste mais significativas. Quantos testes já fizemos e quantos faltam fazer? A taxa de aprovação está aumentando ou diminuindo? Precisamos de gráficos e tabelas para ver rapidamente o status do seu teste. Isso vem de relatórios precisos dos casos de teste e seus resultados. Se ainda não documentamos isso, é o momento de configurar um sistema para rastrear nossa próxima rodada de testes.

Uma vez que o processo de documentação esteja pronto, podemos considerar o conteúdo dos testes, começando com o nível de detalhe que eles devem conter.

TEMA 2 – PREPARAÇÃO DO AMBIENTE

A preparação do ambiente de teste é uma fase do processo de testes e garante o funcionamento correto do *software* a ser colocado em produção.

Este ambiente deve ser similar ao ambiente de produção, incluindo desde configuração de *hardware*, sistemas operacionais, variáveis de ambiente, SGBDs, aplicativos e quaisquer outras dependências relevantes.

Ao prepararmos o ambiente de teste, é importante ter em mente que ele deve ser seguro, confiável e acessível para os testadores, além de fornecer um ambiente controlado para a realização dos testes. O ambiente de teste também deve ser facilmente configurável para permitir testes de diferentes cenários e casos de uso.

Para preparar o ambiente de teste, é necessário estabelecer uma documentação clara e detalhada da configuração, incluindo informações sobre *hardware*, sistema operacional, banco de dados e outras dependências. Outro aspecto importante encontra-se em termos uma equipe dedicada para manter e gerenciar o ambiente de teste.

Podemos testar nas máquinas do desenvolvedor, onde o novo código foi escrito segundos antes, em sua instância de execução ao vivo disponível para seus usuários ou em vários locais intermediários, como áreas de teste dedicadas. O uso de VMs e contêineres, bem como a definição de infraestrutura como código, permite criar ambientes consistentes. Escolher o caminho certo é vital para criar um processo de teste reprodutível e bem-sucedido, e garantir que comecemos os testes no lugar certo (Amey, 2022).

Nosso teste também deve ter as forças conflitantes de ser sistemático e rigoroso, tentando cuidadosamente cada possibilidade, ao mesmo tempo em que é impulsionado pela imaginação e curiosidade, e guiado pelo *feedback* sobre erros anteriores. A especificação é o nosso guia e precisamos testá-la minuciosamente, mas também aprimorá-la à medida que avança.

Exploraremos como atingir esses objetivos conflitantes simultaneamente, tais como:

- entendendo os diferentes níveis de teste;
- definindo casos de teste;
- etapas do teste prescritivo e descritivo;
- avaliando diferentes ambientes de teste;
- definindo a versão, configuração e ambiente corretos;
- realizando testes sistemáticos;
- testes no ciclo de lançamento;
- usando curiosidade e *feedback*.

Os testes unitários (unitários) devem ser escritos para abranger funções individuais, e os testes de integração são necessários para conduzir módulos ou serviços separados. Eles são conhecidos como *Testes Funcionais de Caixa Branca* e são geralmente mais fáceis de escrever porque a funcionalidade em teste é mais simples. No entanto, existe uma classe de problemas que só aparece durante os testes do sistema e, como seus clientes executam todo o sistema em conjunto, esses são os testes mais realistas e completos para realizarmos.

2.1 Avaliando diferentes ambientes de teste

Para testarmos um sistema, precisamos de um ambiente de teste. O teste unitário e alguns testes de integração podem ser executados em módulos e componentes individuais, mas para realizar o teste de sistema, precisamos de um sistema para testar, como o nome sugere. Isso não deve ocorrer em máquinas de desenvolvimento onde os desenvolvedores podem estar fazendo mudanças constantes e não pode ser o ambiente de produção, quando é tarde demais para evitar os danos causados pelos *bugs*. Precisamos de um ambiente de teste entre o ambiente de desenvolvimento e o ambiente de produção para realizar seus testes. Se não o tivermos, configurá-lo é sua primeira tarefa antes de fazer qualquer teste.

O ambiente de teste pode ser uma instalação em branco que ativamos conforme necessário, selecionando o código mais recente. Como alternativa, podemos ter uma área de preparação ou ambiente beta em constante execução. Ambas as abordagens têm benefícios diferentes, considerados a seguir.

2.2 Usando ambientes de teste temporários

Este quadro mostra as vantagens e desvantagens dos ambientes de teste temporários:

Quadro 1 – Vantagens e desvantagens dos ambientes de teste temporário

Vantagens	Desvantagens
● Garantido para estar executando o código mais recente; ● Sempre comece a partir de um estado conhecido; ● Sem manutenção contínua; ● Sem custos de funcionamento contínuos; ● Fácil de testar várias alterações em paralelo.	● Todos os usuários precisam ser capazes de criar o ambiente; ● Precisa de configuração para testes fora do padrão; ● Nenhum uso histórico real.

Fonte: elaborado com base em Amey, 2022.

Com ambientes de teste temporários, iniciamos de um estado conhecido, o qual tem a garantia de executar o código mais recente. Isso é ótimo para desenvolvimento quando podemos experimentar uma nova mudança em comparação com o comportamento antigo, e isso significa que não há manutenção contínua ou custos operacionais quando o sistema não está em uso. O melhor de tudo é que muitos usuários diferentes podem criar ambientes, para que todos possam fazer várias alterações simultaneamente. Isso torna esse arranjo necessário para o desenvolvimento. Os testadores também podem testar as alterações isoladamente quando são novas e têm maior probabilidade de apresentar problemas. Depois de terem passado no teste inicial, as alterações podem ser combinadas.

Por outro lado, todos os usuários precisam ser capazes de criar esse ambiente; não há nenhum ali pronto para ser usado. Isso é mais fácil para desenvolvedores que estão mais acostumados com as ferramentas e passam a maior parte do tempo trabalhando no sistema. Para usuários que usam o sistema de teste apenas como parte de seu trabalho, como proprietários de produtos ou equipes de suporte ou documentação, configurar um novo ambiente pode ser um desafio maior, dependendo da simplicidade e confiabilidade do processo. Idealmente, use a infraestrutura como código para especificar e criar o ambiente da mesma forma sempre que for necessário. Deve haver o menor número

possível de etapas para os usuários, para que seja mais rápido e fácil, com menos lugares onde possa dar errado.

Como o sistema é sempre recém-criado, devemos definir todas as configurações não padrão necessárias para o teste. Podemos simular isso gerando dados de teste, mas pode não revelar problemas genuínos.

Quadro 2 – Vantagens e desvantagens de um ambiente de testes permanente

Vantagens	Desvantagens
● Rápido de usar para testes curtos; ● Mais fácil de monitorar; ● Pode encontrar problemas devido à execução de longo prazo; ● Acumular dados históricos.	● Problemas afetam muitos usuários simultaneamente; ● Requer manutenção; ● Pode conter dados incorretos de compilações de desenvolvimento; ● Exigir atualizações.

Fonte: elaborado com base em Amey, 2022.

As áreas de preparação são ótimas para testes curtos. Eles são executados continuamente, para que possamos acessar rapidamente o recurso necessário e experimentar o caso de teste. Isso é útil para proprietários de produtos que estão testando novos recursos ou testadores respondendo a uma pergunta rápida. Podemos configurar o monitoramento que corresponda ao sistema ativo na área de preparação para relatar erros ou problemas do sistema. Os erros de preparação facilitam a detecção de problemas que aparecem com o tempo, como vazamentos de memória ou uso excessivo de disco. Eles também acumulam naturalmente dados históricos, como a configuração criada em versões antigas que foram migradas. Isso significa que obtemos alguns testes em segundo plano gratuitamente, embora seja difícil avaliar sua extensão e cobertura.

Por outro lado, as áreas de preparação requerem atualizações e manutenção. Podemos automatizar as atualizações, mas as falhas precisam ser investigadas e triadas, e isso pode ser uma tarefa difícil e impopular entre a equipe. As áreas de preparação também podem gerar erros espúrios devido a dados incorretos gravados pelo código de desenvolvimento. Se uma versão anterior escreveu dados incorretos, devemos encontrar e corrigir o código que deu errado e remover ou corrigir os dados errados. Isso pode criar problemas sutis, e já vi problemas aparecerem mais de um ano após o *bug* inicial. Por exemplo, uma nova versão do código pode adicionar verificações adicionais que

rejeitam dados inválidos, enquanto antes não havia sintomas visíveis. Um ambiente de teste temporário com novos dados para cada instalação evita esses problemas desinteressantes.

Finalmente, como muitos usuários compartilham ambientes de preparação, qualquer interrupção causa problemas para muitas pessoas. Podemos realizar testes destrutivos sem dar um aviso e potencialmente atrapalhar outras pessoas.

Ambientes de teste temporários e áreas de preparação são úteis para tarefas diferentes e é importante o uso de ambos. Os ambientes temporários são úteis para os desenvolvedores trabalharem e as áreas de preparação são úteis para testes mais realistas ou testes rápidos, como aqueles executados pelos proprietários do produto.

Depois de selecionar o ambiente de teste e executá-lo com sucesso, devemos garantir que está na versão correta com a configuração correta.

2.3 Definindo a versão, configuração e ambiente corretos

Uma das maiores perdas de tempo ao testar é usar a versão ou configuração errada. O teste pode ocorrer rapidamente quando tudo está funcionando conforme o planejado. Assim que algo der errado, paramos para investigar, o que deve ser reservado para *bugs* genuínos. Qualquer coisa que possamos consertar sozinhos, devemos fazê-lo primeiro.

Antes de executar a versão correta, seu produto deve ter o controle de versão adequado. Se sua equipe de desenvolvimento permitir que testemos qualquer código que esteja no repositório, essa é a primeira coisa que precisa mudar. A equipe de teste precisa usar uma compilação estável e numerada. Isso não é necessário para testes exploratórios e testes de nível inferior, como testes de componente e integração, que podem ser executados em cada compilação e usar apenas um número de compilação. No entanto, quando executamos os testes funcionais abrangentes, precisará saber qual versão do código estava executando.

Em implantações totalmente de Integração-Contínua/Entrega-Contínua (CI/CD), em que todas as alterações são testadas e em seguida implementadas no ambiente, o controle de versão se torna muito mais fluido, mas mesmo aí podemos especificar compilações individuais ou marcar compilações específicas para executar testes. Essa *tag* pode ser tão singular quanto “compilação noturna

1.432”, mas quando projetamos e executamos testes de sistema, usando todo o sistema junto, obtemos números de compilação da equipe de desenvolvimento como um requisito difícil.

Essas compilações devem estar disponíveis todos os dias durante um projeto para garantir que as alterações sejam realizadas regularmente, mas não com tanta frequência que o sistema em teste esteja em constante alteração. Para testes de sistema, compilações sendo produzidas mais de uma vez por dia são muito frequentes. Da mesma forma, se obtivermos apenas uma compilação por semana, isso nos permite que muito tempo seja desperdiçado entre a introdução de defeitos e o teste e descoberta deles. Uma compilação a cada dia mantém tudo em movimento. Se as atualizações enviarem automaticamente o novo código para o ambiente de teste, melhor ainda.

Os processos da equipe de desenvolvimento variam enormemente, geralmente em função do tamanho da equipe.

Se tivermos as compilações numeradas, atualizadas a cada poucos dias e um ambiente para executá-las, podemos preparar o sistema para ser testado. Quais versões precisamos para habilitar esse recurso? Esses detalhes devem ser rastreados na especificação do recurso. Muitas vezes, haverá uma máquina óbvia que devemos atualizar, mas há outras de que precisamos?

É provável que qualquer sistema complexo tenha várias compilações independentes com diferentes números de compilação; portanto, além da versão óbvia que pretendemos testar, é necessário conhecer todas as suas dependências. Quais outros sistemas precisamos atualizar para usar o recurso com sucesso? O desenvolvedor responsável deve fornecer uma lista completa, portanto, certifique-se de ter tudo no lugar. Esses detalhes não fazem parte da especificação do recurso principal, mas devem fazer parte das notas que descrevem a implementação.

Em seguida, devemos verificar se a configuração habilitou o novo recurso. Embora os desenvolvedores tenham possibilitado ativá-lo, ele está realmente ativado? Novamente, esta é uma maneira fácil de perder tempo. Certifique-se de que todos os requisitos estejam documentados na especificação do recurso e, em seguida, verifique e habilite-os em seu ambiente de teste antes de começar.

O uso de sinalizadores de recurso é uma ótima maneira de controlar o teste e a distribuição de recursos. Eles permitem ativar um recurso para um subconjunto de usuários ou apenas em determinadas situações. Embora úteis,

eles adicionam outra configuração que verificaremos como testador. Se estamos nos perguntando por que não vemos nosso novo recurso funcionando, devemos verificar primeiro os sinalizadores de recurso.

Quando tivermos todas as versões corretas em execução, com todas as dependências e configurações necessárias habilitadas, o teste sistemático poderá começar. A equipe de teste nem sempre recebe especificações de recursos, e as que existem não são suficientemente detalhadas. Quase sempre cabe aos testadores fazer mais perguntas e preencher as lacunas na descrição. Vimos três etapas importantes para conseguir isso.

Usando o teste exploratório, mapeamos o comportamento de um recurso, considerando as várias abordagens: teste funcional, tratamento de erros, usabilidade, segurança, capacidade de manutenção e teste não funcional. O teste exploratório também permite que encontremos rapidamente problemas que bloqueariam testes mais detalhados.

Em seguida, escrevemos uma especificação de recurso para listar a funcionalidade detalhada. Essas especificações têm um estilo particular: declarações independentes e testáveis que descrevem o comportamento, mas não a implementação.

No entanto, não podemos escrever a especificação sozinho. Informações de pelo menos um proprietário do produto e um desenvolvedor líder. Uma reunião de revisão oficial deve passar por todos os requisitos, por sua vez, para descobrir quaisquer objeções ou preocupações.

A transformação da especificação em casos de teste exige que consideremos muitos outros cenários possíveis, o que forma a habilidade de testar e como projetar planos completos de testes funcionais com rapidez e sucesso.

TEMA 3 – TESTE DE INTEGRAÇÃO

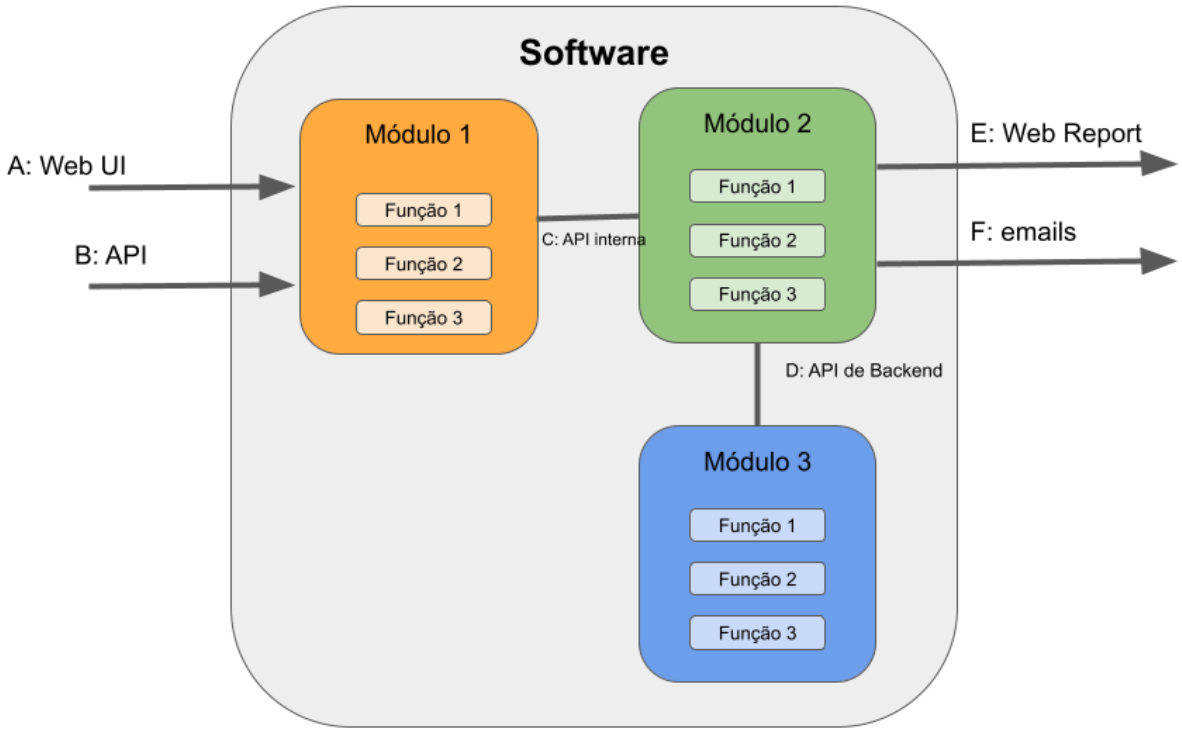
Um teste de integração tem por finalidade a verificação dos diferentes componentes ou módulos de uma aplicação de *software* com o intuito de validarmos e verificarmos se estão trabalhando de forma sincronizada e esperada.

Componentes e módulos podem muito bem funcionar corretamente de forma isolada, mas, ao serem integrados, podem gerar problemas ou defeitos

inesperados. Logo, nossos testes de integração permitem a identificação e correção de tais problemas antes que entreguemos o produto ao nosso cliente.

Considere a seguinte arquitetura simplificada de um sistema de exemplo, conforme mostrado na Figura 3. Ele apresenta duas entradas voltadas para o usuário: uma interface da Web e uma API, identificadas como *A* e *B* no lado esquerdo do diagrama, respectivamente. Essas requisições são processadas pelo Módulo 1, que tem diversas funções, antes de serem passadas por uma API interna, denominada *C*, para o Módulo 2. O Módulo 2 também compreende diversas funções. Ele se comunica com o Módulo 3 usando uma API de *back-end*, rotulada como *D*. O Módulo 2 também apresenta duas interfaces externas somente leitura: um relatório da *web*, rotulado como *E*, e e-mails automatizados, rotulados como *F*:

Figura 3 – Diagrama de um sistema de exemplo, mostrando os testes unitários, integração e sistema



Fonte: elaborado com base em Desikan, 2007.

Os testes unitários estão no nível de teste mais baixo, o qual quebra o código nas menores partes testáveis. Isso significa testar funções e classes individuais para garantir que respondam corretamente às entradas.

O teste de integração isola um módulo específico. Conforme destacado na Figura 3, para testar o Módulo 1, precisamos controlar suas três interfaces: A, B e C. A interface da *web* e a API são voltadas para o público e provavelmente serão bem documentadas e compreendidas. A saída do Módulo 1 é a API interna, C. Para testar isso, devemos imitar o comportamento do Módulo 2 construindo um *stub* ou um programa simulado. *Stubs* são pedaços simples de código que retornam valores fixos — por exemplo, sempre indicando sucesso. Eles estão sem estado e codificados, portanto, estão intimamente associados aos testes que os utilizam. Eles são mais úteis para verificar o estado do sistema em teste.

Mocks são mais complexos. Eles são configuráveis por meio de alguma interface e possuem uma gama mais ampla de comportamento, tornando-os menos acoplados aos testes que os utilizam. Eles verificam a comunicação com o sistema em teste e a interface entre eles.

Em nosso exemplo, precisaríamos de um *stub* ou *mock* na interface C para garantir que o Módulo 1 envie as mensagens corretas. Então, podemos ignorar completamente o comportamento do Módulo 2 e do Módulo 3; eles nem precisam estar em execução. Isso é especialmente útil em um grande projeto para desacoplar os requisitos. Se tivermos um novo recurso e a alteração for concluída no Módulo 1, mas não no Módulo 2, ainda não poderemos testá-lo completamente. No entanto, ao adicioná-lo ao programa simulado na interface C, conseguimos realizar testes de integração no Módulo 1 isoladamente.

Por fim, realizamos os testes de sistema em todo o sistema como um todo. Então, ignoramos as interfaces internas C e D, e testamos apenas as entradas nas interfaces A e B e as saídas nas interfaces E e F. Seus testes devem se concentrar apenas nessas interfaces, embora os testes possam ser informados pelo conteúdo das interfaces C e D quando projetamos testes de caixa branca.

O teste do sistema está intimamente relacionado ao teste de ponta a ponta (*end-to-end* – E2E), que também testa as interfaces externas de um produto. O teste de ponta a ponta é um subconjunto do teste do sistema, com foco específico no fluxo pelos diferentes estados do seu produto e na integridade dos dados durante cada estado. O teste do sistema é mais amplo e abrange usabilidade, capacidade de manutenção, teste de carga e outros assuntos.

No nível mais baixo, os testes unitários devem focar apenas em uma única função em um único módulo do código — neste caso, a função que cria o usuário.

Portanto, o teste chamaria essa função e afirmaria que o usuário está presente no banco de dados com todas as colunas definidas com seus valores padrão corretos.

Este teste precisa que o código e o banco de dados estejam em execução, mas não depende da interface do usuário (UI) ou de quaisquer APIs internas. Mesmo dentro desse teste limitado, precisamos verificar todas as colunas para garantir que estejam preenchidas corretamente, e não apenas verifique a presença do usuário. Mesmo com um teste pequeno e aparentemente simples, é possível deixar lacunas.

O nível intermediário é o teste de integração — por exemplo, APIs internas entre diferentes módulos dentro do código. Um teste de exemplo enviaria uma mensagem de API interna ao banco de dados para criar um usuário. Verifique se o usuário está disponível com seus dados preenchidos corretamente.

Essa mensagem da API deve ser idêntica àquela que o sistema *front-end* envia para o banco de dados. Procure imitar o uso real o mais próximo possível, o que significa atualizar o teste se o comportamento do *front-end* mudar. É necessário mantermos esses testes atualizados ou corre-se o risco de perder *bugs*, pois eles se comportam de maneira diferente do sistema em produção.

Quando o usuário for criado, verifique seu resultado no mesmo nível enviando outro comando com a mesma API. Isso garante que o teste de uma única interface, mas verifica o desempenho de todo o módulo, incluindo banco de dados, processamento e transmissão de dados. Se verificarmos o banco de dados diretamente, por exemplo, como no teste unitário anterior, perderemos *bugs* na API interna.

Por fim, os testes de sistema utilizam todo o sistema de forma realista. Um teste de exemplo criaria um usuário e garantiria que todas as suas informações estivessem visíveis na tela do perfil.

Este teste requer a interface do usuário e todas as etapas internas. Ele depende do funcionamento da interface do usuário, bem como de sua conexão com o *back-end* e o banco de dados. A verificação é executada no mesmo nível novamente, lendo a IU para verificar todo o aplicativo. O teste do sistema é o mais complexo e tem mais dependências; é também o mais lento para executar. No entanto, ele fornece o teste mais completo. Se isso passar, as interfaces internas devem estar funcionando.

O exemplo de teste unitário cobriu uma função de *back-end*, mas também devemos ter testes unitários para o *front-end*, é claro. Por exemplo, verifique se as telas do usuário são idênticas após cada alteração de código que deveria deixá-las intocadas.

Os testes de integração são mais realistas do que os testes unitários porque testam um módulo ou serviço inteiro. Com muitas funções trabalhando juntas, podemos testar suas interações sem exigir que todo o sistema esteja em execução. Os pontos fortes e fracos dessa abordagem são os seguintes:

3.1 Vantagens e desvantagens dos testes de integração

Quadro 3 – Vantagens e desvantagens dos testes de integração

Vantagens	Desvantagens
● Pode desacoplar seções de grandes projetos; ● Combinar testes com unidades funcionais; ● Teste de correspondência com unidades organizacionais; ● As falhas são isoladas; ● Mais simples que os testes de sistema; ● Agnóstico à implementação; ● Não requer um sistema completo.	● Leva tempo para configurar *stubs* e *mocks*; ● Risco de ser irrealista; ● Tome tempo do desenvolvedor; ● Mais complicado que testes unitários; ● Cobertura limitada.

Os testes de integração ajudam a desacoplar diferentes partes dos projetos. Se uma equipe concluiu o trabalho em seu módulo, podemos testar isso isoladamente e ganhar confiança, mesmo que outro trabalho necessário em outro lugar ainda não esteja concluído. Precisamos de mais testes do sistema quando ambas as seções estiverem concluídas, mas os testes podem começar antes disso. Os testes de integração correspondem às unidades funcionais do código, facilitando a identificação de quem é o responsável por quaisquer problemas encontrados. Se uma única equipe possuir esse módulo, essa equipe precisará corrigir os problemas que encontrarmos lá. As falhas são mais rápidas de diagnosticar porque vêm de um único módulo, tornando-as mais fáceis de depurar do que os testes de sistema em que vários módulos trabalham juntos.

Ao contrário dos testes unitários, os testes de integração são independentes da implementação. Podemos refatorar o código completamente,

mas enquanto o comportamento nas interfaces do módulo permanecer inalterado, os testes continuarão passando sem serem atualizados. Os testes de integração também não requerem um sistema completo, tornando-os mais baratos e menos intensivos em recursos.

A desvantagem de não rodar em todo o sistema é que devemos substituir *stubs* ou programas fictícios nas interfaces para outras partes do sistema. Esses levam tempo para serem escritos, precisam ser configurados e mantidos, e precisam ser atualizados para refletir com precisão o comportamento dos outros módulos. Isso leva tempo de desenvolvimento e, mesmo assim, os *stubs* podem ser irrealistas. Isso pode invalidar o teste, o que é sempre um risco com testes de integração.

Um teste de integração pode falhar porque um módulo específico retorna uma resposta inválida para uma entrada específica. No entanto, se ele nunca receber essa entrada no sistema real, o *bug* nunca será acionado. Os testes de integração podem encontrar esse problema, que deve ser corrigido caso uma futura alteração de código o revele. No entanto, esse não é um *bug* realista, portanto, é um resultado falso-positivo. O teste do sistema evita esses falsos positivos realizando apenas testes realistas.

Os testes de integração são muito mais complicados do que os testes unitários para executar e depurar, mas mesmo com essa complexidade extra, ainda há classes de problemas que eles não conseguem encontrar, como problemas entre os módulos. Dois módulos podem funcionar individualmente e passar em seus respectivos testes de integração, mas ainda podem falhar quando executados juntos. Por exemplo, se um módulo contar usuários incluindo usuários com deficiência, enquanto outro contar usuários excluindo usuários com deficiência, seus respectivos totais e comportamentos não corresponderão. Só podemos encontrar essa classe de *bug* com o teste do sistema.

O teste de integração pode ser útil em sistemas difíceis de configurar e testar como um todo. Ainda assim, eles exigem esforço para serem executados e mantidos, embora faltem a simplicidade dos testes unitários e a cobertura abrangente dos testes de sistema.

TEMA 4 – TESTE DE REGRESSÃO

Precisamos escolher quando executar esses testes diferentes em seu ciclo de lançamento. Podemos selecionar um subconjunto de seus testes

automatizados para executar em cada alteração de produto como parte de um pipeline de CI/CD, separado do teste de novos recursos. Esses testes de CI/CD são verificações vitais para evitar que *bugs* e problemas sejam executados ao vivo, especialmente se forem definidos para bloquear o processo de lançamento. No entanto, esses testes têm requisitos estritos:

- Velocidade: eles devem correr rápido o suficiente para não atrasar excessivamente os lançamentos;
- Confiabilidade: eles devem funcionar de forma consistente e só falham quando há um problema real;
- Cobertura: eles devem cobrir todos os aspectos críticos do seu produto.

Existe uma troca natural entre velocidade e cobertura: quanto mais testamos, mais lento ele fica, então precisamos avaliar cuidadosamente os testes a serem incluídos. Esses testes também precisam ser altamente confiáveis, pois são executados com muita frequência e podem atrasar os lançamentos. Os testes unitários são ideais aqui, pois têm menos dependências, mas precisamos de alguns testes de sistema para garantir que o aplicativo geral ainda funcione.

No entanto, os testes de CI/CD são apenas um local no ciclo de lançamento em que podemos executar testes. Os planos de teste podem ser categorizados em quatro níveis, conforme mostrado na Figura 4.

Figura 4 – Subconjuntos de casos de teste

Fonte: elaborado com base em Aniche, 2022.

A camada base são testes abrangentes de novos recursos, cobrindo tudo e qualquer coisa que possamos imaginar. Alguns desses testes podem ser executados apenas uma vez quando um novo recurso é adicionado. Por exemplo, se tivermos uma lista suspensa de países com 200 opções, poderá tentar todas, exceto uma vez. Se a Albânia funcionar, não há razão para suspeitar que a Armênia não funcionará, mas devemos tentar antes de ir ao ar pela primeira vez.

No próximo nível estão os testes de regressão. Eles são quase tão extensos quanto os testes abrangentes, mas incluem apenas os testes que desejamos executar quando esse recurso for modificado. Embora não haja alterações de código, não precisamos executar todos esses testes.

Depois dos testes de regressão, temos os testes manuais/noturnos. Eles têm um limite de tempo – por exemplo, 12 horas para uma execução noturna – portanto, precisam ser priorizados com cuidado. Eles encontram problemas que podem afetar usuários ativos e seriam pequenas interrupções, mas abrangem casos que consomem muito tempo para serem incluídos nos testes de CI/CD.

Por fim, os testes de CI/CD abrangem rapidamente as funções críticas do seu programa, fornecendo confiança após cada alteração de código.

Esses níveis, juntamente com exemplos, estão resumidos neste quadro:

Quadro 4 – Exemplos de diferentes níveis de plano de teste

Nível	Quando é executado	Exemplo de Teste
Testes abrangentes	Uma vez, quando um novo recurso é adicionado	Teste se os usuários podem ser criados com todas as configurações de país possíveis
Testes de regressão	Sempre que esse recurso for alterado	Teste se os usuários podem ser criados com todos os idiomas localizados e uma seleção de outros
Testes manuais/noturnos	À noite ou sob demanda	Teste se todas as páginas estão localizadas para cada idioma

Testes CI/CD	Após cada alteração de código	Teste se uma tela está localizada para cada idioma

Fonte: elaborado com base em Aniche, 2022.

Aqui, nos concentramos em escrever testes abrangentes que cobrem muitas eventualidades. Com uma grande biblioteca de testes para seu sistema, podemos escolher qual promover para as execuções de teste mais curtas. O processo de repetição de testes após alterações em um programa é chamado de *teste de regressão*.

O teste de regressão utiliza casos de teste existentes para verificar se as alterações feitas não produziram novas falhas e não tiveram efeitos colaterais não intencionais. Em outras palavras, o objetivo é garantir que as partes de um sistema revisado que não foram alteradas ainda funcionem como antes das alterações. A maneira mais simples de fazer isso é realizar os testes existentes na nova versão do programa.

Para que os casos de teste existentes sejam úteis para testes de regressão, eles devem ser repetíveis. Isso significa que os casos de teste manuais devem ser suficientemente bem documentados. Os casos de teste usados em testes de regressão serão usados regularmente e com frequência, portanto, são predestinados para automação de teste. A automação de casos de teste de regressão é extremamente útil, pois garante repetibilidade precisa e simultaneamente reduz o custo de cada repetição de teste.

Como estamos verificando se a funcionalidade existente não foi prejudicada (involuntariamente), basicamente precisamos refazer todos os testes que cobrem essa funcionalidade preexistente. Se muito poucos (ou nenhum) testes forem automatizados e, portanto, tivermos que realizar testes de regressão manualmente, será necessário selecionar o menor subconjunto possível dos testes manuais. Para selecionar os casos de teste apropriados, devemos analisar cuidadosamente as especificações de teste para ver quais casos de teste estão relacionados a quais funcionalidades preexistentes e quais se referem à nova funcionalidade modificada.

Se automatizamos casos de teste, a estratégia mais simples é simplesmente reexecutar todos eles para a nova versão do produto. Os casos de teste que retornam um resultado “aprovado” indicam um componente/recurso

inalterado. Isso pode ocorrer porque a alteração necessária não foi feita ou porque os casos de teste “antigos” não foram formulados com precisão suficiente para cobrir a funcionalidade modificada. Em ambos os casos, os casos de teste correspondentes precisam ser modificados para garantir que reajam à nova funcionalidade.

Casos de teste que retornam um resultado “falha” indicam funcionalidade modificada. Se isso incluir recursos que não foram sinalizados para alteração, uma falha é garantida, pois indica que, ao contrário do planejado, o recurso correspondente foi alterado. Como essas modificações não intencionais são reveladas pelos casos de teste “antigos”, nenhuma alteração adicional é necessária.

Para recursos que precisam ser alterados, os casos de teste também precisam ser adaptados para que cubram a nova funcionalidade. O resultado de tudo isso é um conjunto de testes de regressão que verificam as alterações planejadas na funcionalidade. Os casos de teste que cobrem funções completamente novas ainda não fazem parte desta suíte e devem ser desenvolvidos separadamente.

Na prática, o teste de regressão completo que executa todos os testes existentes geralmente é muito caro e leva muito tempo, especialmente quando (como mencionado acima) estão envolvidos testes manuais. Portanto, estamos procurando critérios que nos ajudem a decidir quais casos de teste herdados podem ser ignorados sem perder muita informação. Como sempre em um contexto de teste, isso requer um compromisso entre minimizar custos e aceitar riscos de negócios.

A seguir, estão estratégias comuns para selecionar casos de teste:

- repita apenas os testes que receberam alta prioridade no cronograma de teste;
- deixe de fora casos especiais para testes funcionais;
- limite o teste a determinadas configurações específicas (por exemplo, teste apenas a versão em inglês, teste apenas para um sistema operacional específico e similares);
- limite o teste a componentes ou níveis de teste específicos.

As regras listadas aqui se aplicam principalmente ao teste do sistema. Em níveis de teste inferiores, os critérios de teste de regressão podem ser baseados

na documentação de *design* (como a hierarquia de classes) ou em informações de caixa branca. O teste de regressão utiliza casos de teste existentes para verificar se as alterações feitas não produziram novas falhas e não tiveram efeitos colaterais não intencionais. Em outras palavras, o objetivo é garantir que as partes de um sistema revisado que não foram alteradas ainda funcionem como antes das alterações.

A maneira mais simples de fazer isso é realizar os testes existentes na nova versão do programa. Para que os casos de teste existentes sejam úteis para testes de regressão, eles devem ser repetíveis. Isso significa que os casos de teste manuais devem ser suficientemente bem documentados. Os casos de teste usados em testes de regressão serão usados regularmente e com frequência e, portanto, predestinados para automação de teste. A automação de casos de teste de regressão é extremamente útil, pois garante repetibilidade precisa e simultaneamente reduz o custo de cada repetição de teste.

Se muito poucos (ou nenhum) testes forem automatizados e, portanto, tivermos que realizar testes de regressão manualmente, será necessário selecionar o menor subconjunto possível dos testes manuais. Para selecionar os casos de teste apropriados, devemos analisar cuidadosamente as especificações de teste para ver quais casos de teste estão relacionados a quais funcionalidades preexistentes e quais se referem à nova funcionalidade modificada.

Se automatizamos casos de teste, a estratégia mais simples é simplesmente reexecutar todos eles para a nova versão do produto. Os casos de teste que retornam um resultado “aprovado” indicam um componente/recurso inalterado. Isso pode ocorrer porque a alteração necessária não foi feita ou porque os casos de teste “antigos” não foram formulados com precisão suficiente para cobrir a funcionalidade modificada. Em ambos os casos, os casos de teste correspondentes precisam ser modificados para garantir que reajam à nova funcionalidade.

Casos de teste que retornam um resultado “falha” indicam funcionalidade modificada. Se isso incluir recursos que não foram sinalizados para alteração, uma falha é garantida, pois indica que, ao contrário do planejado, o recurso correspondente foi alterado. Como essas modificações não intencionais são reveladas pelos casos de teste “antigos”, nenhuma alteração adicional é necessária.

Para recursos que precisam ser alterados, os casos de teste também precisam ser adaptados para que cubram a nova funcionalidade. O resultado de tudo isso é um conjunto de testes de regressão que verificam as alterações planejadas na funcionalidade. Os casos de teste que cobrem funções completamente novas ainda não fazem parte desta suíte e devem ser desenvolvidos separadamente.

Na prática, o teste de regressão completo que executa todos os testes existentes geralmente é muito caro e leva muito tempo, especialmente quando (como mencionado acima) estão envolvidos testes manuais. Portanto, estamos procurando critérios que nos ajudem a decidir quais casos de teste herdados podem ser ignorados sem perder muita informação. Como sempre, em um contexto de teste, isso requer um compromisso entre minimizar custos e aceitar riscos de negócios.

A seguir, estão estratégias comuns para selecionar casos de teste:

- Repita apenas os testes que receberam alta prioridade no cronograma de teste.
- Deixe de fora casos especiais para testes funcionais.
- Limite o teste a determinadas configurações específicas (por exemplo, teste apenas a versão em inglês, teste apenas para um sistema operacional específico e similares).
- Limite o teste a componentes ou níveis de teste específicos. As regras listadas aqui se aplicam principalmente ao teste do sistema.

Em níveis de teste inferiores, os critérios de teste de regressão podem ser baseados na documentação de *design* (como a hierarquia de classes) ou em informações de caixa branca.

Resumindo um pouco, um teste de regressão é responsável pela verificação das possíveis alterações e mais recentes em um código com o foco de identificarmos se afetam indevidamente recursos ou funcionalidades que antes funcionavam. Ou seja, funcionalidades que foram implementadas e testadas com sucesso anteriormente, categorizadas como uma verificação que garante que um novo desenvolvimento não tenha “regredido” ou prejudicado o que já existia.

TEMA 5 – TESTE DE SISTEMA

Após a conclusão do teste de integração, o próximo nível de teste é o teste do sistema. Este nível verifica se o sistema completo e integrado realmente atende aos requisitos especificados. Aqui, também podemos nos perguntar por que essa etapa é necessária após o teste bem-sucedido de componente e integração.

Os testes de baixo nível verificam as especificações técnicas do ponto de vista do fabricante do *software*. Em contraste, o teste do sistema vê o sistema do ponto de vista do cliente e do usuário final. Os testadores do sistema verificam se os requisitos especificados foram implementados de forma completa e adequada.

Muitas funções e atributos do sistema resultam da interação dos componentes do sistema e, portanto, só podem ser observados e testados em nível de sistema.

> Na fase de teste de sistema, o objetivo é executar o sistema sob ponto de vista de seu usuário final, varrendo as funcionalidades em busca de falhas em relação aos objetivos originais. Os testes são executados em condições similares – de ambiente, interfaces sistêmicas e massas de dados – àquelas que um usuário utilizará no seu dia a dia de manipulação do sistema. De acordo com a política de uma organização, podem ser utilizadas condições reais de ambiente, interfaces sistêmicas e massas de dados.

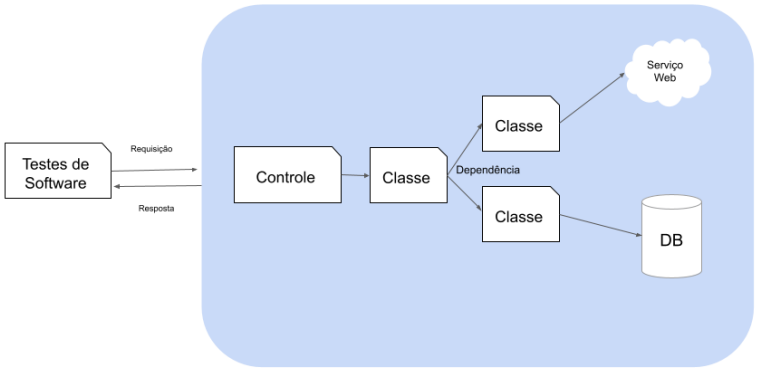
Os testes de sistema fazem parte do topo da pirâmide de testes e são responsáveis por testes que simulam o uso de um sistema por um usuário/*stakeholder* real. Muito conhecido como de testes ponta-a-ponta (*end-to-end*) ou até como testes de interfaces. São testes dispendiosos que demandam muito esforço para a implementação e mais tempo para execução.

Como exemplo, temos o *framework* Selenium, que automatiza testes para sistemas na *web*. Ele permite a criação de pequenos robôs que abrem e simulam o preenchimento de formulários, cliques, testes de resultados, entre outros.

Para obter uma visão mais realista do *software* e, assim, realizar também os testes de forma mais realista, devemos executar todo o sistema de *software* com todos os seus bancos de dados, aplicativos de *front-end* e outros componentes. Quando testamos o sistema em sua totalidade, em vez de testar pequenas partes do sistema isoladamente, estamos fazendo um teste de sistema Figura 5. Não nos importamos como o sistema funciona por dentro; não importa se foi desenvolvido em Java ou Ruby, ou se utiliza um banco de dados

relacional. Só nos importamos que, dada a entrada X, o sistema forneça a saída Y (Aniche, 2022).

Figura 5 – Teste do sistema: nosso objetivo é testar todo o sistema e seus componentes



Fonte: elaborado com base em Aniche, 2022.

A vantagem óbvia do teste de sistema é o quão realistas são os testes. Nossos clientes finais não executarão o método identityExtremes() isoladamente. Em vez disso, eles visitarão uma página da *web*, enviarão um formulário e verão os resultados. Os testes de sistema exercitam o sistema dessa maneira precisa. Quanto mais realistas forem os testes (ou seja, quando os testes executam ações semelhantes ao usuário final), mais confiantes podemos estar sobre todo o sistema.

Os testes de sistema geralmente são lentos em comparação com os testes unitários. Imagine tudo o que um teste de sistema deve fazer, incluindo iniciar e executar todo o sistema com todos os seus componentes. O teste também precisa interagir com o aplicativo real e as ações podem levar alguns segundos. Imagine um teste que inicia um *container* com uma aplicação *web* e outro *container* com um banco de dados. Em seguida, ele envia uma solicitação HTTP para um serviço da *web* exposto por esse aplicativo da *web*. Este serviço da *web* recupera dados do banco de dados e grava uma resposta JSON para o teste. Obviamente, isso leva mais tempo do que executar um teste unitário simples, que praticamente não tem dependências.

Os testes de sistema também são mais difíceis de escrever. Alguns dos componentes (como bancos de dados) podem exigir uma configuração complexa antes de serem usados em um cenário de teste. Pense em conectar, autenticar e garantir que o banco de dados tenha todos os dados exigidos por

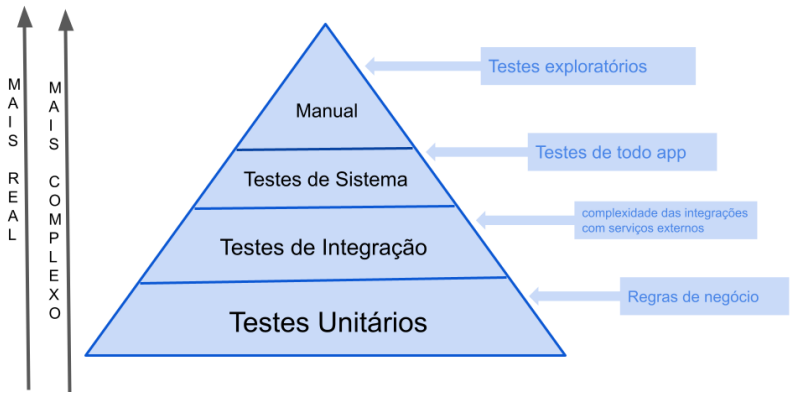
esse caso de teste. O código adicional é necessário apenas para automatizar os testes.

Os testes do sistema são mais propensos a falhas. Um teste *flaky* apresenta comportamento errático: se o executarmos, ele pode passar ou falhar para a mesma configuração. Imagine um teste de sistema que exercite um aplicativo da *web*. Depois que o testador clica em um botão, a solicitação HTTP POST para o aplicativo da *web* leva meio segundo a mais do que o normal (devido a pequenas variações que muitas vezes não controlamos em cenários da vida real). O teste não espera isso e, portanto, falha. O teste é executado novamente, o aplicativo da *web* leva o tempo normal para responder e o teste passa. Muitas incertezas em um teste de sistema podem levar a um comportamento inesperado.

Com uma compreensão clara dos diferentes níveis de teste e seus benefícios, temos que decidir se devemos investir mais em teste unitário ou teste de sistema e determinar quais componentes devem ser testados por meio de teste unitário e quais componentes devem ser testados por meio de teste de sistema. Uma decisão errada pode ter um impacto considerável na qualidade do sistema: um nível errado pode custar muitos recursos e não encontrar *bugs* suficientes.

Alguns desenvolvedores preferem o teste unitário a outros níveis de teste. Isso não significa que esses desenvolvedores não façam integração ou teste de sistema; mas sempre que possível, eles empurram o teste para o nível de teste unitário. Uma pirâmide é frequentemente usada para ilustrar essa ideia, conforme mostrado na Figura 6. O tamanho da fatia na pirâmide representa o número relativo de testes a serem realizados em cada nível de teste.

Figura 6 – Pirâmide de teste



Fonte: elaborado com base em Aniche, 2022.

O teste unitário está na base da pirâmide e tem a maior área. Isso significa que os desenvolvedores que seguem esse esquema favorecem o teste unitário (ou seja, escrevem mais testes unitários). Subindo no diagrama, o próximo nível é o teste de integração. A área é menor, indicando que, na prática, esses desenvolvedores escrevem menos testes de integração do que testes unitários. Dado o esforço extra que os testes de integração exigem, os desenvolvedores escrevem testes apenas para as integrações de que precisam. O diagrama mostra que esses desenvolvedores favorecem menos os testes de sistema do que os testes de integração e têm ainda menos testes manuais.

Diante de toda hierarquia de testes, este tema tem por foco explicar qual o objetivo dos testes de sistemas. É uma etapa no processo de desenvolvimento de *software* importante, porque verifica se o *software* realmente está completo e atende aos requisitos funcionais. O importante é que ele entregue aquilo que foi definido para seu bom funcionamento (requisitos). Abordam a verificação da integração, da funcionalidade e da performance, além de garantir a qualidade antes da entrega ao cliente.

FINALIZANDO

A visão sistêmica da abordagem de testes, validações e verificações de *software* nos proporciona exatamente a ideia de início, meio e fim do processo como um todo.

Um *software*, sistema ou aplicação deve ser visto como um conjunto de partes que interagem entre si com o objetivo final de entregar ao cliente uma solução tecnológica para resolução de um problema. Logo, nosso *software* deve ser pensado e projetado no momento do desenvolvimento, mas também no momento dos testes, validações e verificações.

A etapa de preparação do ambiente de testes, a visão dos testes de integração, regressão e do sistema (ponto a ponto) nos trazem conforto nas entregas, pois temos uma visão sistêmica do funcionamento total do novo produto ou de pequenas implementações feitas em um produto já existente.

A pior situação para um usuário de uma aplicação é quando há alguma modificação (release ou versão) e aquilo que funcionava maravilhosamente simplesmente deixa de funcionar ou tem algo que piora a funcionalidade.

Entregas de novas implementações sobre produtos já existentes são mais desafiadoras por conta disso. Nosso cliente já está acostumado dentro de uma

questão de funcionalidade, usabilidade, performance, entre outros requisitos não funcionais.

Em linhas gerais, mergulhar na visão sistêmica de testes nos faz pensarmos o quão importante e planejadas devem ser tais mudanças, bem como toda sua carga de testes.

REFERÊNCIAS

AMEY, S. **Software Test Design**. Packt Publishing, 2022.

ANICHE, M. **Effective Software Testing**. Manning Publications, 2022.

ASSOCIATION FOR COMPUTING MACHINERY. Disponível em: <https://www.acm.org/>. Acesso em: 12 mar. 2023.

BECK, K. **Test-driven development**. Addison-Wesley, 2003.

BROWN, E. **Web development with Node and Express**. Sebastopol: O'Reilly Media, Inc., 2014.

DESIKAN, S. **Software Testing**: Principles and Practices. O'Reilly Media, Inc., 2007.

GARCIA, B. **Mastering Software Testing with JUnit 5**. Packt, 2017.

HAMBLING, B. et al. **Software testing**: an ISTQB-BCS certified tester foundation guide. 3. ed. Swindon: BCS Learning & Development Ltd., 2015.

IBM100. **The Origins of Computer Science**. Disponível em: <https://www.ibm.com/ibm/history/ibm100/us/en/icons/compsci/>. Acesso em: 12 mar. 2023.

IEEE – Institute of Electrical and Electronic Engineers. **The world's largest technical professional organization dedicated to advancing technology for the benefit of humanity**. Disponível em: <https://www.ieee.org/>. Acesso em: 12 mar. 2023.

ISO – International Organization for Standardization. **ISO 8402**:1994. Disponível em: <https://www.iso.org/standard/20115.html>. Acesso em: 12 mar. 2023.

_____. **ISO 9001 and related standards — Quality management**. Disponível em: <https://www.iso.org/iso-9001-quality-management.html>. Acesso em: 12 mar. 2023.

_____. **ISO/IEC 25010**:2011. Disponível em: <https://www.iso.org/standard/35733.html>. Acesso em: 12 mar. 2023.

_____. **ISO/IEC/IEEE 24765**:2017. Disponível em: <https://www.iso.org/standard/71952.html>. Acesso em: 12 mar. 2023.

ISTQB – International Software Testing Qualifications Board. Certified Tester **Foundation Level (CTFL)**. Disponível em: <https://www.istqb.org/certifications/certified-tester-foundation-level>. Acesso em: 12 mar. 2023.

JUNIT. **Teste de Unidade**. Disponível em: <http://junit.wikidot.com/>. Acesso em: 12 mar. 2023.

LAPORTE, C.; APRIL, A. **Software quality assurance**. Wiley: IEEE Press, 2018.

LATINO, R. **Root cause analysis**: improving performance for bottom-line results. CRC Press, 2011.

LEWIS, W. E. **Software testing and continuous quality improvement**. 3. ed. Boca Raton: Taylor & Francis Group, LLC., 2009.

MOHAN, G. **Full Stack Testing**. O'Reilly Media, Inc., 2022.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011.

SOMMERVILLE, I. **Engenharia de software**. São Paulo: Pearson Education do Brasil, 2018.